Extended Model Driven Architecture to B Method (short version)

Ammar Aljer, Philippe Devienne

Extended Model Driven Architecture to B Method

Ammar Aljer
Faculty of Electrical and Electronic Engineering
Aleppo University, Aleppo, Syria
Tel: +963 988 975 621
ammar.aljer@lifl.fr

Philippe Devienne
Lille Computer Science Laboratory (LIFL)
University of Science and Technology of Lille (USTL)
59655 Villeneuve d'Ascq, France
Tel: +33 3 28 77 85 78
Fax: +33 3 28 77 85 37
philippe.devienne@lifl.fr

Extended Model Driven Architecture to B Method

ABSTRACT

Model Driven Architecture (MDA) design approach proposes to separate design into two stages: implementation independent stage then an implementation-dependent one. This improves the reusability, the reusability, the standability, the maintainability, etc. Here we show how MDA can be augmented using a formal refinement approach: B method. Doing so enables to gradually refine the development from the abstract specification to the executing implementation; furthermore it permits to prove the coherence between components in low levels even if they are implemented in different technologies.

Keywords: MDA, B method, Co-design Refinement, Embedded System, VHDL

1. Introduction

As computer performance improves and human-built systems augment, there are continuous efforts to employ suitable Computer Aided design tools that are able to develop such complex systems. A common attitude between designers in different technologies is to use more abstract design levels that enable designer to concentrate, at first, on the most important requirements of the system.

In hardware domain, many tools are produced to develop higher levels than printed circuits or RTL (register transfer level). VHDL (IEEE 1076) is emerged on 1987. it permits to represent a complete hardware system. It became the dominant in Hardware modelling. VerilogSystem is standardised in 2005 to manage abstract level of hardware system.

In software area, number of OOP languages has emerged. They give more facilities to treat complex system than procedural languages. An implementation-independent tool, UML (unified modelling language), use graphical diagrams to gather common aspects of OOP Languages using. An object oriented system is made up of interaction components. Each component (object) has its own local state and provides operations on that state. In Object oriented design process, Designer concentrates more on precising classes (abstraction of real objects) and the relationships between these classes. MDA (model driven architecture) was launched by the OMG (Object Management Group) in 2001. It proposes to separate the design into two stages: implementation-independent stage then an implementation-dependent one. "The transition between these stages of development should, ideally, be seamless, with compatible notation used at each stage. Moving to the next stage involves refining the previous stage by adding details to exiting object classes and devising new classes to provide additional functionality. As information is concealed within objects, detailed design decision about the representation of data can be delayed until the system is implemented."[8].

Another important aspect of nowadays systems is the interference between different technologies. Most systems consist of different cooperating sub-systems where some functionality may migrate from one technology to another in further versions of the system.

In our project, that is illustrated in figure 1, we improved MDA approach in three main aspects:

1. Smoothing transfer from the abstract specification of the system into the implementation with a proven refinement from each level to the next and the more deterministic one.
2. Formal notation of the complete system in the abstract levels
3. Formal projection of components that are implemented in hardware technology.

Our approach (that joints the advantages of MDA and B method) permits to obtain many advantages:

1. The possibility to obtain a correct-by-design system
2. Increase the reusability: when a modification is necessary, we preserve all design levels that are more abstract than the level where modification is occurred.
3. The possibility of migration between technologies in low levels without reproving the complete system if the immigration preserves the logical behaviour captured in the formal projection.

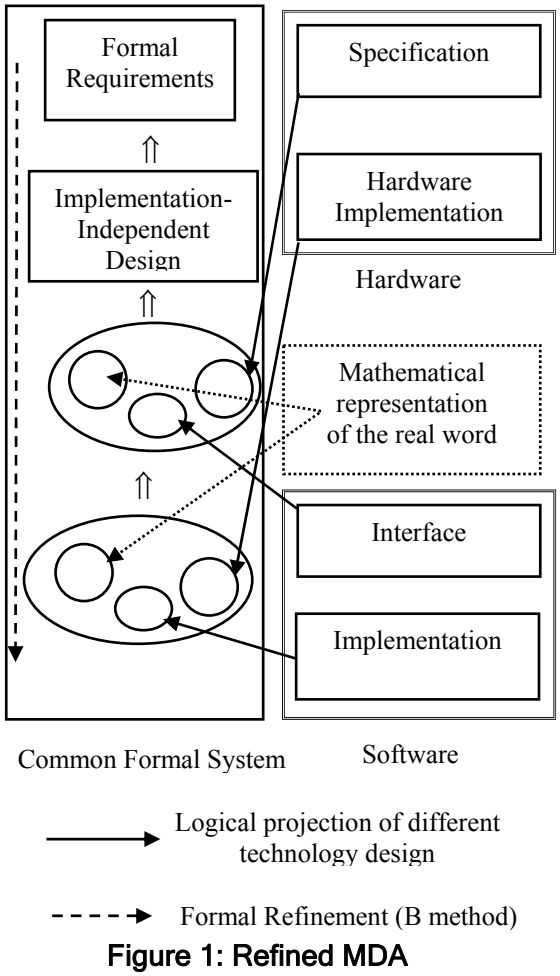


Figure 1: Refined MDA

Figure 1 shows that the first step is to formally specify the requirements. This step may be achieved during an iterative process where new requirements do not contradict with the previous ones. This step may be followed by one or more steps to design the main components of the wanted system independently of the implementation technology. Using the formal refinement of B, components in each step is proven to be coherent and refine the previous step. Designers in each community may use their own development tools and techniques to implement a part of the system. A formal representation of the implementation of the different technologies is traced to prove the compatibility and the implantation- independent architecture. If necessary, the system may be proven in coexisting with mathematical representation of parts of the real environments such as physical laws, external systems, etc.

2. MDA and BHDL

Most Co-design verification methods depend on Co-simulation of two or more types of components that are designed by different technologies. Each research community tries to extend design stages to include more abstract levels. Fortunately, we can observe many common properties in the research result of these different communities. It is quite interesting to compare them and to show that they could be prefigured and structured within a model driven architecture. In this paper, we focus on B and VHDL.

B method [1] is known in software engineering as a formal method to specify and to finely develop the specification towards an executable program basing on set theory and first order logic notation. During the software development in B method, many versions of the same component may be found. The first and the most abstract one is the *abstract machine* where client needs are declared. Then, the following versions should be more concretes. They should describe more and more "how" we obtain the needed specifications. These versions are called *refinements* except the last one where there is no more possible refinement. This deterministic version is called *implementation*. B tools generate the necessary proof obligations to verify the coherence of each component and correctness of the development. Furthermore, B tools help to execute these proofs.

VHDL [2] is a dominant in Hardware description. The designer may use two descriptions of a circuit; *ENTITY* and *ARCHITECTURE*. In the first one, the interface of the circuit with its environment is specified and in the second one the internal structure of the circuit is detailed. Many standard and private libraries and packages may be used to facilitate the design.

As it is defined in its web site, **VGUI** is a Graphical User Interface for Hardware Diagrams. It may be considered as a simple component description tool. VGUI may be used to create generic interconnected boxes. Each box may be decomposed hierarchically into sub-boxes and so on. The boxes and the connections are typed.

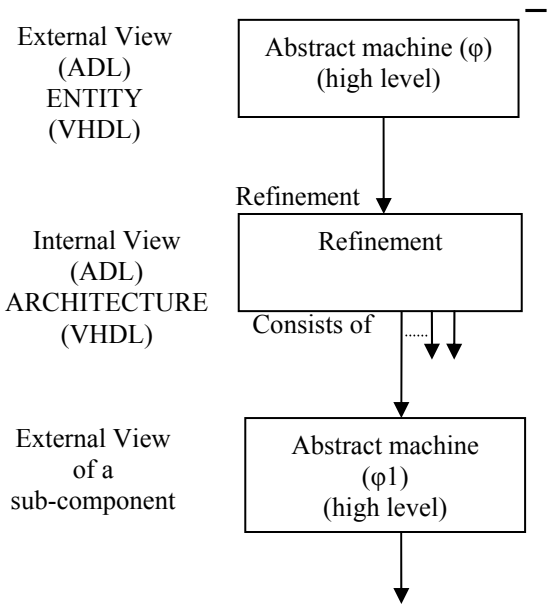


Figure2: Common Aspects between ADL, HDL and B.

The principle of **BHDL** is to make use of the common properties between B, ADL and HDL in order to use a common formal iterance language. This will facilitate the verification of design correctness since the early steps of co-design. Fortunately, B method has its own mathematical notation that can be used during all development steps. The correctness of a system described by B language may be "proven" by many tools as AtelierB, BToolkit , B-For-Free and RODIN [3].

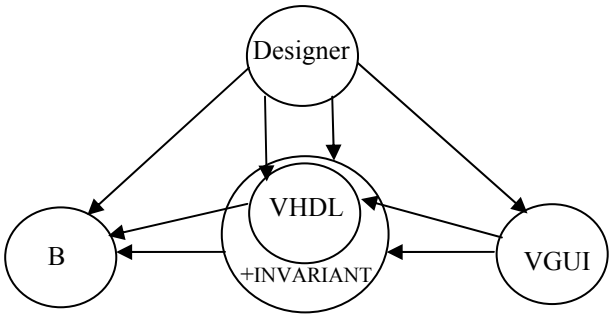


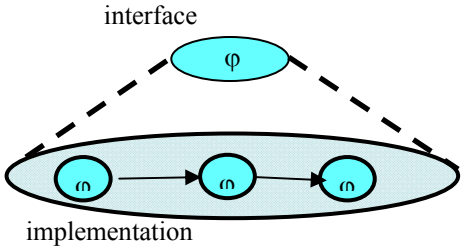
Figure3: Principle of BHDL.

In the initial version of VGUI we could not attached logic properties to a component (a Box). In cooperation with VGUI developer, we added the possibility to attach logic property to each box.

From VGUI interface the main structure of the system is created. Then, two different notations are generated: VHDL and B. The produced B code contains the main features of VHDL one. After that, design may be separated in relation to the technologic choices. VGUI has a limited expression power, but our VHDL-B translator can handle any VHDL code.

3. BHDL : B ↔ VHDL

VHDL is a language for describing the structural, physical and behavioral characteristics of digital systems. As we have said, two VHDL basic components are used to represent the hierarchy; the Entity and the ARCHITECTURE. The first one defines the interface of the system (or of a component). It specifies the connection ports of the component and the type of transmitted signals. While the second represents the internal structure (or behavior) of the system (or of a component). Each Architecture is attached to one Entity and it may contain recursively one or more Entitys. This structure looks similar to extern-view and intern-view in ADL, procedure call and procedure implementation in imperative language .etc. Also in B method two basic components excite: the Abstract machine and the Refinement. The first one is usually used to precise the specifications of the component; the interface variables, the internal variables, the invariant relation between them and the pre and post conditions of the necessary operations. The second component may refine an abstract machine; that means it precise partly how the operations may be implemented. The Refinement component may be, in his turn, refined recursively by more deterministic Refinements. The last refinement step, when the behavior becomes completely deterministic, is called the implementation. B tools may prove the consistency of each component and the refinement relation. In our project each Entity is translated by an Abstract machine and each Architecture by a refinement. The ports are declared as Variables and the port typing as Invariant. Furthermore we enhanced the VHDL notation with logical properties. These properties are injected in B Invariant. The connection between subcomponents of the Refinement should guarantee the Invariant specified in the abstract machine (see figure4).



$\varphi_1 \wedge \varphi_2 \wedge \varphi_n \wedge \texttt{links} \Rightarrow \varphi$

Figure4: BHDL Refinement.

3.1 Hierarchy

In VHDL, the transition from an Entity into a corresponding Architecture is usually performed in one step. In BHDL, this may be finely performed by many steps or levels. We may consider the refinement of a component in BHDL as a replacement by other components. Also we may refine a component by another one which has the same structure and links but with more strict logic property. In all cases the refinement is performed towards lower levels where the behavior of the system becomes more deterministic.

The principal relation between the interface (external view) and its refinement (or between two levels of refinement) is:

$$\text{Connection}(\varphi_1, \varphi_2, \ldots \varphi_n) \Rightarrow \varphi$$

which means that the logical connection between the properties of the sub-components should satisfied the properties indicated in the abstract machine that represents the Entity.

2.2. Compositionality and Invariant

Let us consider the following simple example for illustrating captures of multiple mathematical views and reliability.

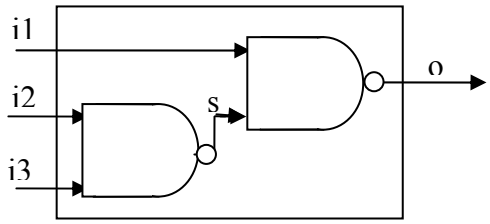


Figure 5: Structure of Comp1 component.

Figure (5) shows a system that contains two Nand components. The modified version of VGUI allow to draw a similar connected boxes and to precise the logic properties and the internal structure of each box. Then VHDL$^+$ and B code is generated.

VGUI generated the following VHDL$^+$ code for this example:

```
STRUCTURE comp1 OF comp
SIGNAL s
BEGIN
  gate1 : nand  PORT MAP (i1,s,o)
  gate2 : nand  PORT MAP (i2,i3,s)
END
ENTITY nand
  PORT x, y : IN std_logic
       z : OUT std_logic
 -- z = nand (x,y) B specification
 END
```

3.2. Specification Languages

As B is used in this example as formal specification language, PSL is an "add-on" language for Hardware description languages that has recently been standardized by the IEEE in 2005. PSL standard is based upon IBM's "Sugar" language, which was developed and validated at IBM Labs for many years before IBM donated the language to Accellera for standardization. PSL works alongside a design written in VHDL, Verilog or SystemVerilog. But in future it may be extended to work with other languages. Properties written in PSL may be embedded within the HDL code as comments or may be placed in a separated file alongside the HDL code. PSL includes multiple abstraction layers for assertion types ranging from low-level Boolean and Temporal to higher-level Modeling and Verification. Formally, PSL is structured into four layers: the Boolean, Temporal, Verification and Modeling layers. At its lowest-level, PSL uses references to signals, variables and values that exist in the design's conventional HDL description. Sugar used CTL (Computation Tree Logic) formalism to express properties for model checking. But the finally the underling semantic foundation was migrated from CTL to LTL (Linear-Time Temporal Logic) because the latter is considered more accessible to a wider audience and it is more suitable for simulation. The temporal operators of the foundation language provide syntactic sugaring on the top of LTL operators. These temporal operators include:

- Always: it holds if its operator holds in every signal cycle.
- Never: it holds if its operand fails to hold in every signal cycle.
- Next: it holds if its operand holds in the cycle that in the immediately follows.
- Until: it holds if the property at its left-hand holds in every cycle from the current cycle up until the next cycle in which the property at its right-hand holds.
- Before: it holds if the left-hand operand holds at least once between the current cycle and the next time the right-hand operand holds.

3.3. Fault tolerance

The usual development in B method goes from the abstract requirement to the concrete execution. During the development, the behavior becomes more and more deterministic. In spite of that, BHDL can takes in account the possibility to describe a fault scenario. Here we describe the ideal system with the behavior of the ideal variables in the abstract machine, then, by Refinement, we inject the possible fault. This fault is declared using false variables. Then, we propose the correction step for the false variables. At the end, we prove that the corrected values of the false variables respect the INVARIANT of the initial ones. The additional variables and the correction operations are the cost of trust behavior of the system.

3.4. Dependency Relation

BHDL project can make use of B tools to verify the dependence between an output and an input. In Refinement components, each connection produces an independency relation between two variables. Two types of connections may be noticed; the connection between the sub-components and the

intern wires and the connection between sub-components and outer ports.

The direction of the dependency is related to the signal direction. As we see, this relation recursively depends on the lower levels. As Refinement (architecture) can see only the abstract machines (ENTITYs) of its sub-components. So that, as the Refinement can not see the Refinements of its own sub-components, it cannot see their dependency relation (see figure 6). One solution is to modify the Invariant of each Abstract machine where dependency relation is declared. To facilitate the modification we write a part the invariant of the abstract machine in an independent file that may be easily modified by the refinement.

We defined a transitive relation "Depend" on the ensemble PORTS with one direction. This relation should be defined on variables attached to the instances of the interne components not to the generic form of them so we add new variables for each instance to define the dependency relation. For example, we shall write the dependency relation for the following component.

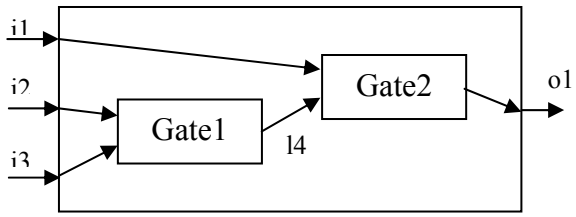


Figure 6: Dependency Relation.

All these modification of the INVARIANT are applied at refinement level where we can see the subcomponents. But we need this information at the abstract machine level because we need to know the dependency relation in a higher level where this component (or abstract machine) is included, in its turn, as subcomponent. The abstract machine of the right part of figure (7) is used as a sub-component in the refinement of the left part.

This dependency relation has been use to check fan-out property. In digital circuits, fan-out defines the maximum number of digital inputs that the output of a single logic gate can feed. The value of the fan-out is a big impact on test and debugging.

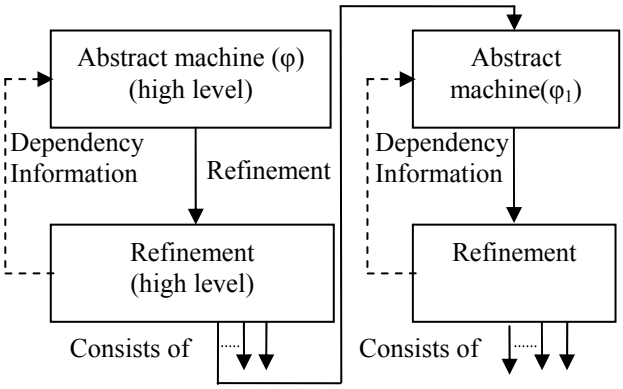


Figure 7: Dependency Information Transfer.

4. AFCIM

The French project AFCIM (Formal Architectures for Conception and Maintenance of Embedded Systems) coordinated by Philippe Devienne (LIFL) is a collaborative research between four French universities and institutes (LIFC, LIFL, Heudiasyc, INRETS).

The global architecture of the AFCIM project is:

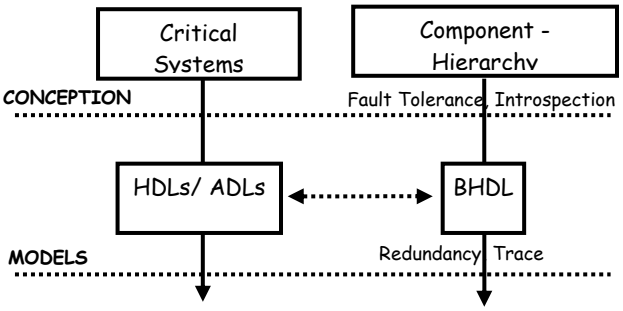


Figure 8: AFCIM

From a general Model Driven Architecture (ie the common part of specific description languages like ADL, HDL...), we add formal annotations and specifications according to the requirements or the fault scenarios that we want to handle. All the tools used in our platform are freely used and distributed (Rodin, Eclipse, Antlr, …). This research first conducted into the AFCIM project (LIFL, INRETS, HEUDIASYC Lab) will be now supported within a PCSI project (Zero Defect Systems) between Lille University, Aleppo University and Annaba University.

References:

[1] J.-R. Abrial, The B-Book: Assigning Programs to Meanings, Cambridge University Press, UK, 1996.

[2] Y. Herve, VHDL-AMS – Applications et enjeux industriels, Duand, France, 2002.

[3] RODIN : http://www.event-b.org/platform.html

[4] Flaviu Cristian, "Understanding Fault-Tolerant Distributed Systems", ACM, February 1991, 34(2): 56-78

[5] Terence Parr, *The definitive ANTLR Reference,* 384 pages, May 2007

[6] *Ammar Aljer, Co-design and refinement in B,* Ph.D. Thesis, 2004.

[7] D. Garlan, "Formal Modeling and Analysis of Software Architecture: Components, Connectors and Events", Springer-Verlag, Sep 2003.

[8] I. Sommerville," Software Engineering", Pearson, 2007

[9] DOULOS, "PSL Golden Reference guide",Book, 2005.